CS406 – Parallel Computing
FUTOSHIKI HW3

Oguzhan Ilter
19584
May 2020

1 Introduction

The aim of this assignment is to design and implement an algorithm that can solve efficiently (in terms of run-time and memory consumption) a large number of Futoshiki Puzzle concurrently. In the implementation, there should be parallelization with CUDA library. The CUDA kernel will solve every Futoshiki Puzzle brutally.

In Futoshiki puzzles, the objective of the game is to discover the digits hidden inside the board's cells; each cell is filled with a digit between 1 and the board's size. On each row and column each digit appears exactly once. Some digits may be given at the start. Inequality constraints are initially specified between some of the squares, such that one must be higher or lower than its neighbor. These constraints must be satisfied in order to complete the puzzle.

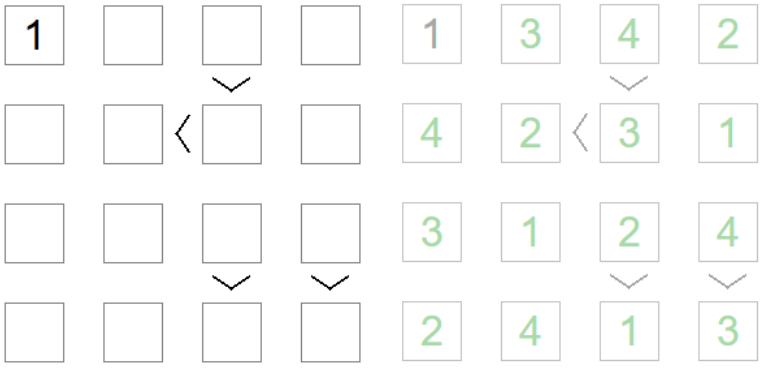

Figure 1: Example of Futoshiki Puzzle and its answer

In the following sections, the implementation steps along with the problems and their solutions are going to be discussed. In addition, the execution time of the algorithm will be provided.

2 Implementation

In this homework, the implementation was converted to iterative one due to the limitation of GPU on recursive algorithms. Therefore, a new algorithm has been created which does not include forward checking. The new algorithm checks every value for a grid and if the value is valid for the time being, the next grid is taken into the process. If the algorithm reaches a dead-end it goes back to last valid state.

The new solver does not contain any *return* to terminate the process because it has to try possible states until all grids are filled and find a solution. Note that every thread in CUDA is responsible for one input.

Algorithm General Structure of Iterative GPU Solver
Require (*matrix, constraints, number of constraint*)

```
1: FOR Every Grid in Puzzle
2:     IF Grid is not pre-set
3:         FOR Every Possible Value of a Grid
4:             IF Value is Valid
5:                 Go next Grid
6:             ELSE
7:                 Try next possible value
8:         IF Dead-End
9:                 Revert changes on matrix
10:                Go previous valid state
```

Code 1: Pseudocode of puzzle solver

The whole code has three main parts:

I. Input part: read data from txt file and store them into dynamically allocated arrays.
II. Solver part: Copy the data to GPU device, solving multiple puzzles in GPU concurrently copy data to host from the device.
III. Checking part: Check the correctness of the solved puzzles and fill an output txt.

During the development phase, the solver part has been constantly changed in order to find the optimal usage of multiple threads. In my implementation every CUDA thread is responsible for a puzzle. A single thread can solve a puzzle around 0.005 sec. This time has been reduced from 5 sec which was performed by the initial recursive solver algorithm. The usage recursive algorithm

has been discarded because GPU is not compatible with many recursive calls due to its small stack size.

In the solver part, the copy phase has been eased by flatting the inputs matrix, in other words all inputs have been transformed into 1D dynamically allocated arrays. During the development of solver part, one of the versions of the early kernels was not able to solve every matrix at once, due to the large stack size need of the solver algorithm. To overcome this issue, I tried to implement a loop which calls kernel repetitively to solve every puzzle in parts. However, this attempt was failed due to segmentation faults. Therefore, I had to come up with a better iterative algorithm.

The checking part checks every output matrix one by one if their final states are valid or not. In addition to that it creates the output txt with the name *solved.txt.*

The current solver algorithm does not force any thread to occupy its own private memory by creating or copying any data unnecessarily. So that kernel can solve 144000 matrices at once in around 1.114 sec (average of 50 runs). Other timings are as follows:

	Copy HostToDevice	**Solver**	**Copy DeviceToHost**
Time (sec)	0.015227	1.11444	0.004993

Table 1: Timing Average of 50 runs

The profiling results of the algorithm is as follows.

Type	Time(%)	Time	Calls	Avg	Min	Max	Name
GPU activities:	98.48%	1.11814s	1	1.11814s	1.11814s	1.11814s	solver(int*, int*, long*, int*)
	1.12%	12.761ms	4	3.1904ms	896ns	7.9331ms	[CUDA memcpy HtoD]
	0.40%	4.5333ms	1	4.5333ms	4.5333ms	4.5333ms	[CUDA memcpy DtoH]
API calls:	81.92%	1.11815s	3	372.72ms	6.2750us	1.11813s	cudaEventSynchronize
	16.24%	221.65ms	2	110.82ms	1.6970us	221.65ms	cudaEventCreate
	1.39%	18.999ms	5	3.7999ms	84.384us	8.2897ms	cudaMemcpy
	0.30%	4.1237ms	3	1.3746ms	352.32us	2.2971ms	cudaFree
	0.08%	1.0246ms	4	256.15us	172.51us	461.36us	cudaMalloc
	0.03%	390.34us	96	4.0660us	192ns	151.53us	cuDeviceGetAttribute
	0.02%	244.90us	1	244.90us	244.90us	244.90us	cudaLaunchKernel
	0.01%	203.61us	1	203.61us	203.61us	203.61us	cuDeviceTotalMem
	0.01%	79.967us	6	13.327us	4.5740us	48.865us	cudaEventRecord
	0.00%	43.158us	1	43.158us	43.158us	43.158us	cuDeviceGetName
	0.00%	17.433us	3	5.8110us	2.4720us	11.150us	cudaEventElapsedTime
	0.00%	16.701us	1	16.701us	16.701us	16.701us	cudaSetDevice
	0.00%	10.858us	1	10.858us	10.858us	10.858us	cudaDeviceSynchronize
	0.00%	4.5830us	1	4.5830us	4.5830us	4.5830us	cuDeviceGetPCIBusId
	0.00%	3.5530us	3	1.1840us	331ns	2.4720us	cuDeviceGetCount
	0.00%	3.2850us	1	3.2850us	3.2850us	3.2850us	cudaPeekAtLastError
	0.00%	1.1310us	2	565ns	251ns	880ns	cuDeviceGet
	0.00%	358ns	1	358ns	358ns	358ns	cuDeviceGetUuid

Table 2: Profiling of code

As it can be seen from table 2, cudaEventSynchronize takes most of the time due to large number of inputs.

3 Tricks Done for Efficiency

- **Try Next Value if Obviously Wrong**
  It can be easily understood that some values are wrong, like a value that causes number repetition on a column or row. At this state, algorithm does not evaluate further the state. It tries another value on the grid or goes back to previous valid state.

- **Flatten Everything**
  According to the instruction of CUDA library, memory copies are preferred in form of 1D arrays due to high performance and easy to use reasons. Therefore, I converted every data array to 1D arrays so that it increases the copy speed. In addition to that it makes easier the debug process.

- **No Need for Saving the State at Evert Try**
  In the previous solutions, algorithms kept previous state information with a matrix that includes constraint relations and possible values of every grid. This was a very memory consuming operation and was not appropriate for GPU operations. In this implementation, states are stored in an array with the same size as the matrix itself. This array also clarifies which grids are set which are changeable.

4 Compile and Run

I compiled the codes via *gcc 4.7.3 and CUDA 10.0* with command on terminal

nvcc xxxx.cu .

In addition, I tested the profiling:

nvprof ./a.out input.txt

5 Comments

Unfortunately, the algorithm contains many if statements which cause many branching. This issue slows down the process, but I could not find a better iterative algorithm that reduces the branching factor.

Another possible improvement is that the data copy and solving can be done simultaneously. While a chunk of data is copied, the already copied data could have been processed. However, I could not implement this due to bugs. The data should have divided into optimal number of chunks and different indexing mechanisms should be developed. Finding the optimal size of chunks is an empiric issue.

REMARK: The BLOCKSIZE and THREADPERBLOCK should be changed according to number of inputs. Output txt file name is written on console.